PYTHON ADVANCED TOPICS

UNB-GGE PYTHON WORKSHOP SERIES 2

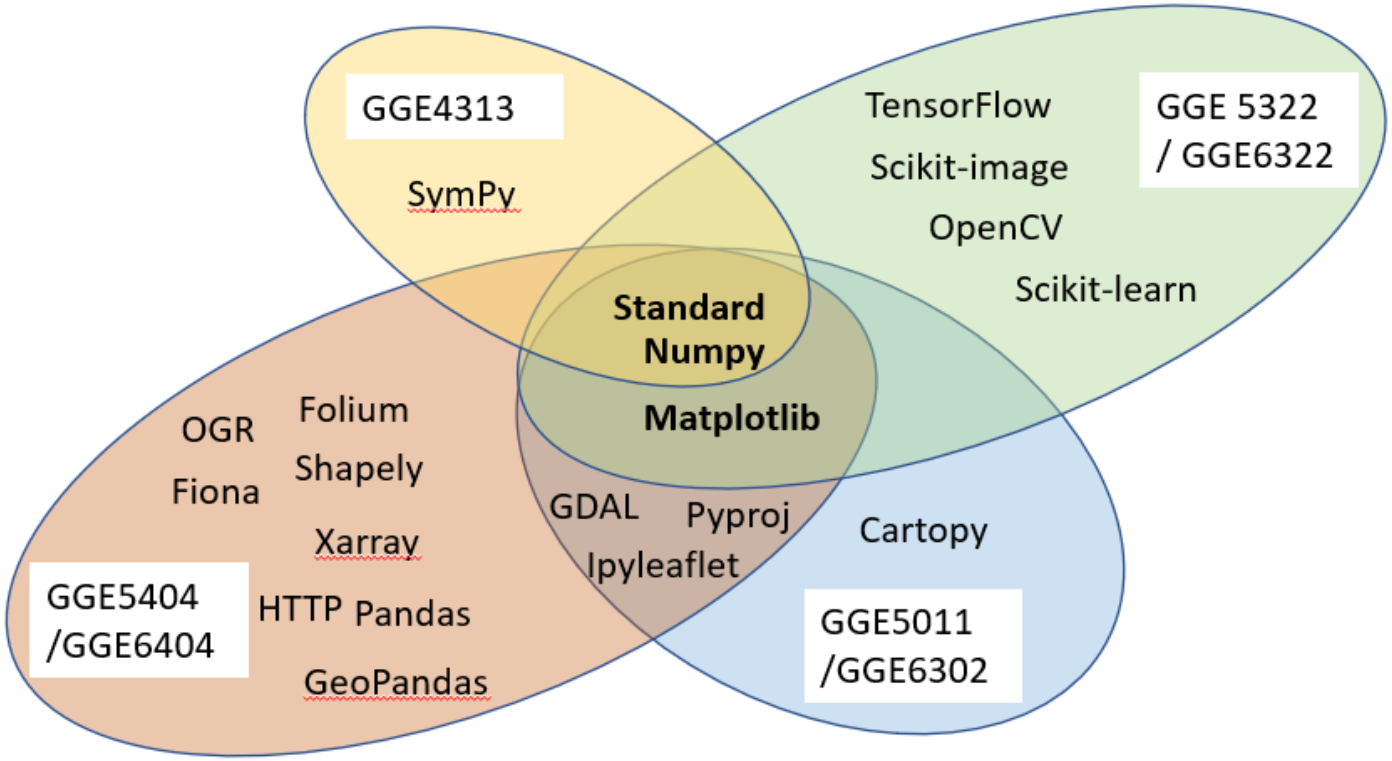
William Liu
Department of Geodesy and Geomatics Engineering
University of New Brunswick
August 2023

Agenda

0. Python Libraries Used in GGE Courses
1. Python Standard Library
2. NumPy - The Python Equivalent of MATLAB
   - 2.1 NumPy ndarray Class
   - 2.2 NumPy ndarray Element Data Types
   - 2.3 Creating arrays (ndarray objects)
   - 2.4 ndarray Properties
   - 2.5 ndarray Methods
   - 2.6 Iteration Over Arrays
3. Matplotlib - Python’s Data Visualization Powerhouse
   - 3.1 Matplotlib Architecture
   - 3.2 Matplotlib Object Hierarchy
   - 3.3 Matplotlib Figure Elements
   - 3.4 Interfaces for Creating Plots
   - 3.5 plt.subplots() Function
   - 3.6 Examples

0. Python Libraries Used in the GGE Courses

Course Code	Course Name
GGE4313	Photogrammetry
GGE5011 / GGE6302	Oceanography, Tides, and Water Levels
GGE5322 / GGE6322	Digital Image Processing / Computer Vision – Algorithm and Software Coding
GGE5404 / GGE6404	Online Spatial Data Handling

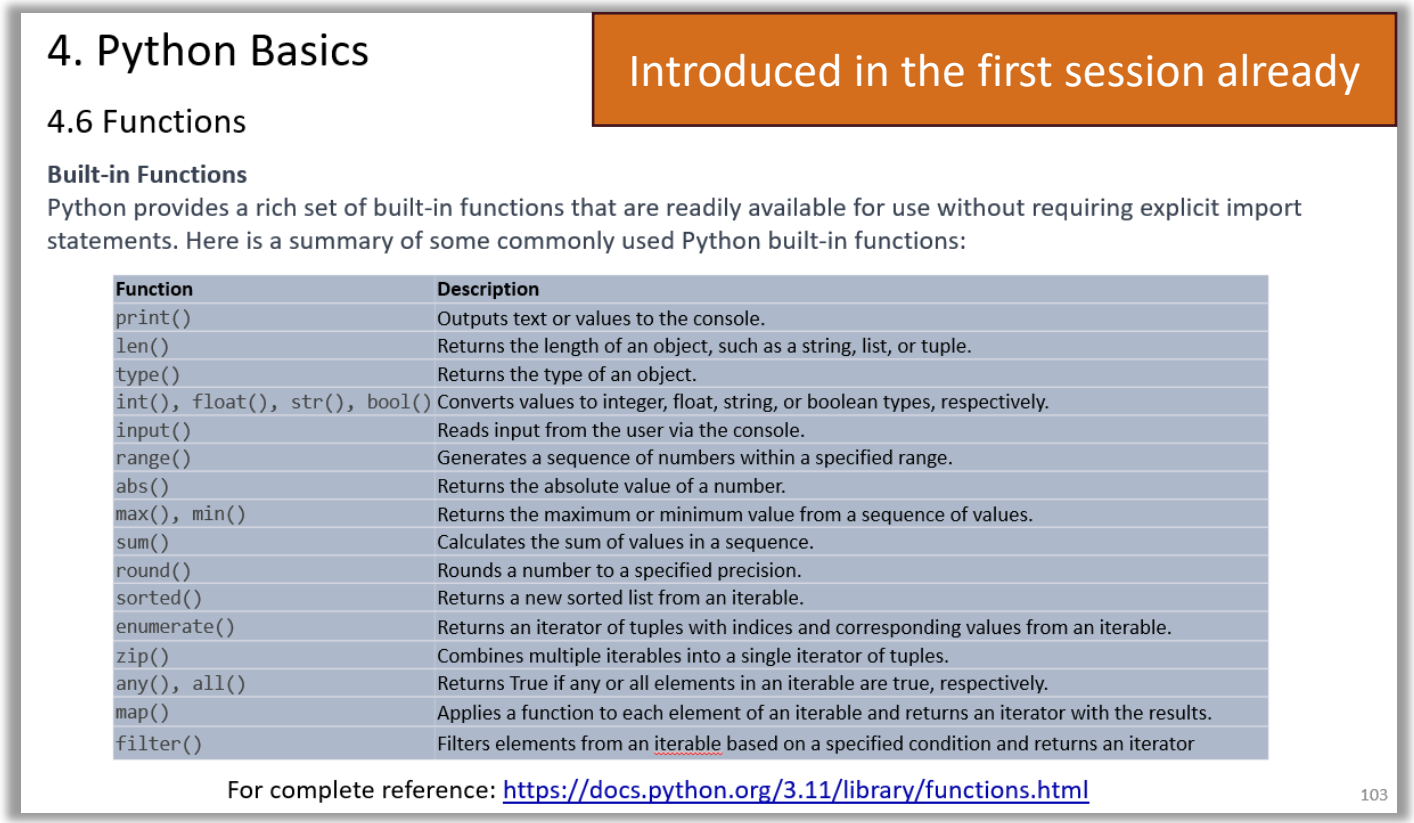


1. Python Standard Library

Python standard library includes built-in functions, and the modules must be imported before use.

Built-in functions:

- They are immediately available without the need to import any external modules. They are always accessible in any Python environment, whether it's an interactive session or a script.
- They perform important tasks like input/output, math calculations, and data manipulation.



4. Python Basics

Introduced in the first session already

4.6 Functions

Built-in Functions

Python provides a rich set of built-in functions that are readily available for use without requiring explicit import statements. Here is a summary of some commonly used Python built-in functions:

Function	Description
print()	Outputs text or values to the console.
len()	Returns the length of an object, such as a string, list, or tuple.
type()	Returns the type of an object.
int(), float(), str(), bool()	Converts values to integer, float, string, or boolean types, respectively.
input()	Reads input from the user via the console.
range()	Generates a sequence of numbers within a specified range.
abs()	Returns the absolute value of a number.
max(), min()	Returns the maximum or minimum value from a sequence of values.
sum()	Calculates the sum of values in a sequence.
round()	Rounds a number to a specified precision.
sorted()	Returns a new sorted list from an iterable.
enumerate()	Returns an iterator of tuples with indices and corresponding values from an iterable.
zip()	Combines multiple iterables into a single iterator of tuples.
any(), all()	Returns True if any or all elements in an iterable are true, respectively.
map()	Applies a function to each element of an iterable and returns an iterator with the results.
filter()	Filters elements from an iterable based on a specified condition and returns an iterator

For complete reference: https://docs.python.org/3.11/library/functions.html

103

1. Python Standard Library

Python standard library includes built-in functions, and the modules must be imported before use.

Modules need explicit import:

- "Python standard library" often means additional specialized tools available in modules and packages.
- It covers various domains and provides many modules, such as file handling, networking, concurrency, etc.

- **os** and **os.path**: Operating system interfaces and file/directory path manipulation.
- **sys**: System-specific parameters and functions.
- **math**: Mathematical functions and operations.
- **random**: Pseudo-random number generators.
- **datetime** and **time**: Date and time manipulation.
- **json** and **pickle**: JSON encoding/decoding and object serialization.
- **collections**: Additional data structures like deque, defaultdict, Counter, etc.
- **io**: Input and output tools.
- **re**: Regular expression operations.
- **string**: String manipulation and formatting.
- **csv**: CSV file reading and writing.
- **urllib**: URL handling and fetching.
- **http**: HTTP client/server operations.
- **socket**: Low-level networking interfaces.
- **argparse**: Command-line argument parsing.
- **logging**: Flexible logging framework.
- **subprocess**: Process creation and management.
- **sqlite3**: SQLite database connectivity.
- **timeit**: Code timing and profiling.
- **gzip** and **zipfile**: Compression and archive manipulation.
- **xml.etree.ElementTree**: XML parsing and creation.
- … https://docs.python.org/3/library/index.html

Study it when you need it!

2. NumPy

The Python Equivalent of MATLAB

2. NumPy

NumPy (Numerical Python) is an open-source Python library for scientific computing and data analysis. A foundation for numerical computations in Python.

- Anaconda has NumPy installed by default
- It supports multi-dimensional arrays and matrices
- Offers a wide range of mathematical functions
- Efficient execution due to C implementation.
- Integrates closely with other libraries like SciPy, Matplotlib, Pandas, and TensorFlow.
- Features:
  - Enables efficient element-wise operations on arrays.
  - Broadcasting allows operations on arrays of different shapes.
  - Provides powerful indexing and slicing capabilities.

.

NumPy Homepage: https://numpy.org/

2. NumPy

2.1 NumPy `ndarray` Class

The NumPy data type `ndarray` (short for "N-dimensional array") is a fundamental concept in the NumPy library. It represents a multidimensional, homogeneous array of elements, which can be of various numerical types.

- **Multidimensional**: The `ndarray` can have any number of dimensions (axes), allowing it to represent scalars (0D), vectors (1D), matrices (2D), and higher-dimensional arrays.
- **Homogeneous**: All elements in an `ndarray` must be of the same data type, ensuring consistent memory usage and efficient computation.
- **Shape**: The shape of an `ndarray` specifies the size of each dimension. It is represented as a tuple of integers, e.g., (rows, columns) for a 2D array.
- **Data Types**: The `ndarray` supports a wide range of data types, such as integers, floating-point numbers, complex numbers, and custom types.
- **Indexing**: Elements in an `ndarray` can be accessed using integer indices along each dimension. Indexing starts at 0.

2. NumPy

2.1 NumPy `ndarray` Class

The NumPy class `ndarray` (short for "N-dimensional array") is a fundamental concept in the NumPy library. It represents a multidimensional, homogeneous array of elements, which can be of various numerical types.

- **Slicing**: You can create sub-arrays (slices) by specifying ranges along each dimension, enabling efficient manipulation and extraction of data.

- **Broadcasting**: NumPy supports broadcasting, which allows operations between arrays of different shapes to be performed seamlessly.

- **Universal Functions (ufuncs):** NumPy provides various element-wise mathematical functions (ufuncs) that can be applied to entire arrays without loops, optimizing computation.

- **Vectorized Operations**: Many operations on `ndarray` objects are performed element-wise, making code concise and efficient.

- **Memory Efficiency**: NumPy's memory layout is efficient and consistent, allowing for optimized memory usage and reduced overhead.

- **Array Creation**: NumPy offers various methods to create `ndarray` objects, including those for initialization, repetition, and stacking.

2. NumPy

2.2 NumPy `ndarray` Element Data Types

NumPy's precise data types are better than Python's built-in types and can optimize memory for numerical data.

Commonly used data types

- Integer
  - signed integers:
    - int8
    - int16
    - int32
    - int64
  - unsigned integers:
    - uint8
    - uint16
    - uint32
    - uint64

Commonly used data types

- Floating-point types
  - float16
  - float32
  - float64
  - float128
  - complex64
  - complex128
  - complex128

Commonly used data types

- Date and time type
  - datetime64
- Date and time difference type
  - timedelta64
- Boolean type
  - bool

2. NumPy

2.3 Creating arrays (`ndarray` objects)

- `numpy.array()`: Create an array from a Python list or tuple.

```
import numpy as np

# Create an array from a Python list or tuple

# Example using default data type inferred from the input data
arr_default = np.array([1, 2, 3])  # Default data type (e.g., int32 or float64)

# Creates a 1D array from a Python list with specified data type if provided
arr_custom = np.array([1, 2, 3], dtype=np.int8)  # Customized data type: int8

print("arr_default:", arr_default)  # Output: arr_default: [1 2 3]
print("arr_custom:", arr_custom)    # Output: arr_custom: [1 2 3]
```

2. NumPy

2.3 Creating arrays (`ndarray` objects)

- `numpy.zeros()`: Create an array filled with zeros of a specified shape

```
import numpy as np

# Create an array filled with zeros of a specified shape (2D, 2 rows by 3 columns)

# Example using default data type
zeros_default = np.zeros((2, 3))  # Default data type: float64

# Example using customized data type
zeros_custom = np.zeros((2, 3), dtype=np.float64)  # Customized data type: float64

print("zeros_default:", zeros_default)  # Output: zeros_default: [[0. 0. 0.]
                                        #                          [0. 0. 0.]]
print("zeros_custom:", zeros_custom)    # Output: zeros_custom: [[0. 0. 0.]
                                        #                        [0. 0. 0.]]
```

2. NumPy

2.3 Creating arrays (`ndarray` objects)

- `numpy.ones()`: Create an array filled with ones of a specified shape

```
import numpy as np

# Create an array filled with ones of a specified shape (2D, 2 rows by 3 columns)

# Example using default data type
ones_default = np.ones((2, 3))  # Default data type: float64

# Example using customized data type
ones_custom = np.ones((2, 3), dtype=np.float32)  # Customized data type: float32

print("ones_default:", ones_default)  # Output: ones_default: [[1. 1. 1.]
                                      #                        [1. 1. 1.]]
print("ones_custom:", ones_custom)    # Output: ones_custom: [[1. 1. 1.]
                                      #                       [1. 1. 1.]]
```

2. NumPy

2.3 Creating arrays (`ndarray` objects)

- `numpy.empty()`: Create an array without initializing its values (contains random data).

```
import numpy as np

# Create an array without initializing its values (contains random data)

# Example using default data type
empty_default = np.empty((2, 2))  # Default data type: platform-dependent

# Example using customized data type
empty_custom = np.empty((2, 2), dtype=np.int64)  # Customized data type: int64

print("empty_default:", empty_default)  # Output: empty_default: [[0.00000000e+000 1.07597970e-282]
                                        #                          [1.28091730e-315 1.63321133e-301]]
print("empty_custom:", empty_custom)    # Output: empty_custom: [[25895968444448860 23925768161198147]
                                        #                        [32370111954616435 29555336418885724]]
```

2. NumPy

2.3 Creating arrays (`ndarray` objects)

- `numpy.arange()`: Create an array with a sequence of values within a specified range.

```
import numpy as np

# Create an array with a sequence of values within a specified range

# Example using default data type
range_default = np.arange(0, 10, 2)  # Default data type: int64

# Example using customized data type
range_custom = np.arange(0, 10, 2, dtype=np.uint8)  # Customized data type: uint8

print("range_default:", range_default)  # Output: range_default: [0 2 4 6 8]
print("range_custom:", range_custom)    # Output: range_custom: [0 2 4 6 8]
```

2. NumPy

2.3 Creating arrays (`ndarray` objects)

- `numpy.linspace()`: Create an array with evenly spaced values over a specified range.

```
import numpy as np

# Create an array with evenly spaced values over a specified range

# Example using default data type
linspace_default = np.linspace(0, 1, 5)  # Default data type: float64

# Example using customized data type
linspace_custom = np.linspace(0, 1, 5, dtype=np.float16)  # Customized data type: float16

print("linspace_default:", linspace_default)  # Output: linspace_default: [0. 0.25 0.5 0.75 1.]
print("linspace_custom:", linspace_custom)    # Output: linspace_custom: [0. 0.25 0.5 0.75 1.]
```

2. NumPy

2.3 Creating arrays (ndarray objects)

- `numpy.logspace()`: Create an array with values evenly spaced on a logarithmic scale.

```
import numpy as np

# Create an array with values evenly spaced on a logarithmic scale

# Example using default data type
logspace_default = np.logspace(1, 3, 4)  # Default data type: float64

# Example using customized data type
logspace_custom = np.logspace(1, 3, 4, dtype=np.float64)  # Customized data type: float64

print("logspace_default:", logspace_default)  # Output: logspace_default: [10. 100. 1000. 10000.]
print("logspace_custom:", logspace_custom)    # Output: logspace_custom: [10. 100. 1000. 10000.]
```

2. NumPy

2.3 Creating arrays (`ndarray` objects)

- `numpy.eye()`: Create a 2D identity matrix (with ones on the diagonal and zeros elsewhere).

```
import numpy as np

# Create a 2D identity matrix or array

# Example using default data type
identity_default = np.eye(3)  # Default data type: float64

# Example using customized data type
identity_custom = np.eye(3, dtype=np.float32)  # Customized data type: float32

print("identity_default:", identity_default)  # Output: identity_default: [[1. 0. 0.]
                                              #                            [0. 1. 0.]
                                              #                            [0. 0. 1.]]
print("identity_custom:", identity_custom)    # Output: identity_custom: [[1. 0. 0.]
                                              #                           [0. 1. 0.]
                                              #                           [0. 0. 1.]]
```

2. NumPy

2.3 Creating arrays (ndarray objects)

- `numpy.full()`: Create an array with a specified constant value.

```
import numpy as np

# Create an array with a specified constant value

# Example using default data type
full_default = np.full((2, 3), 7)  # Default data type: inferred from input (e.g., int32, float64)

# Example using customized data type
full_custom = np.full((2, 3), 7, dtype=np.uint32)  # Customized data type: uint32

print("full_default:", full_default)  # Output: full_default: [[7 7 7]
                                      #                        [7 7 7]]
print("full_custom:", full_custom)    # Output: full_custom: [[7 7 7]
                                      #                       [7 7 7]]
```

2. NumPy

2.3 Creating arrays (ndarray objects)

- `numpy.random.rand()`: Create an array of random values from a uniform distribution over [0, 1).

```
import numpy as np

# Create a 2D array of random values from a uniform distribution over [0, 1)

# Example using default data type
rand_default = np.random.rand(2, 3)  # Default data type: float64

# Example using customized data type
rand_custom = np.random.rand(2, 3).astype(np.float32)  # Customized data type: float32

print("rand_default:", rand_default)  # Output: rand_default: [[0.99497283 0.13255965 0.18791486]
                                      #                        [0.5531434  0.39205125 0.46024962]]
print("rand_custom:", rand_custom)    # Output: rand_custom: [[0.22404945 0.93653077 0.6325018 ]
                                      #                       [0.72025275 0.68015283 0.28603235]]
```

2. NumPy

2.3 Creating arrays (ndarray objects)

- `numpy.random.randn()`: Create an array of random values from a standard normal distribution (mean 0, variance 1).

```
import numpy as np

# Create an array of random values from a standard normal distribution (mean 0, variance 1)

# Example using default data type
randn_default = np.random.randn(2, 3)  # Default data type: float64

# Example using customized data type
randn_custom = np.random.randn(2, 3).astype(np.float64)  # Customized data type: float64

print("randn_default:", randn_default)  # Output: randn_default: [[-0.17782399  0.99609939 -0.13190769]
                                        #                        [-0.14353027  1.50688462 -0.22277069]]
print("randn_custom:", randn_custom)    # Output: randn_custom: [[ 1.14184136 -1.40194685  0.54746621]
                                        #                       [-1.05792355  1.52046108  0.95935307]]
```

2. NumPy

2.3 Creating arrays (ndarray objects)

- `numpy.random.randint()`: Create an array of random integer values within a specified range.

```
import numpy as np

# Create an array of random integer values within a specified range

# Example using default data type
randint_default = np.random.randint(0, 10, (2, 3))  # Default data type: int64

# Example using customized data type
randint_custom = np.random.randint(0, 10, (2, 3), dtype=np.int16)  # Customized data type: int16

print("randint_default:", randint_default)  # Output: randint_default: [[3 3 9]
                                            #                          [8 7 1]]
print("randint_custom:", randint_custom)    # Output: randint_custom: [[2 4 4]
                                            #                         [3 6 7]]
```

2. NumPy

2.3 Creating arrays (`ndarray` objects)

More methods:

- ❑ `numpy.tile()`: Create an array by repeating an existing array
- ❑ `numpy.concatenate()`: Combine multiple arrays along an existing axis
- ❑ `numpy.vstack()`: Stack arrays vertically (along rows)
- ❑ `numpy.hstack()`: Stack arrays horizontally (along columns)
- ❑ `numpy.dstack()`: Stack arrays along the third axis (depth-wise)
- ❑ `numpy.meshgrid()`: Create coordinate grids for 2D plotting
- ❑ `numpy.fromfunction()`: Create an array based on a user-defined function
- ❑ `numpy.fromfile()`: Create an array from data in a binary file
- ❑ `numpy.loadtxt()`: Create an array from data in a text file.

Beware of these methods. Check the documentation when you need them.

2. NumPy

2.4 `ndarray` Properties

`ndarray` properties give crucial information about NumPy arrays' structure, data types, memory usage, and other features. You can access them by appending the property name to the `ndarray` object, like my_array.shape or my_array.dtype. Commonly used `ndarray` properties are:

- `ndarray.shape`: Returns a tuple representing the dimensions of the array. For a 2D array, (rows, columns) is returned.
- `ndarray.ndim`: Returns the number of dimensions (axes) of the array.
- `ndarray.size`: Returns the total number of elements in the array.
- `ndarray.dtype`: Returns the data type of the elements in the array.
- `ndarray.itemsize`: Returns the size (in bytes) of each element in the array.
- `ndarray.nbytes`: Returns the total size (in bytes) of the array.
- `ndarray.data`: Returns a buffer object pointing to the start of the array's data.
- `ndarray.strides`: Returns a tuple of bytes to step in each dimension when traversing the array.
- `ndarray.base`: If the array is a view of another array's data, this property points to the base array; otherwise, it's None.
- `ndarray.flags`: An object containing information about the memory layout of the array.
- `ndarray.T`: Returns the transpose of the array, effectively swapping rows and columns.
- `ndarray.real` and `ndarray.imag`: For complex arrays, these properties return the real and imaginary parts, respectively.
- `ndarray.flat`: A 1-D iterator over the array's elements.
- `ndarray.item`: If the array has only one element, this property returns that element.

2. NumPy

2.4 ndarray Properties

Examples of ndarray properties:

```
import numpy as np

# Example 1: 1D Array
arr_1d = np.array([1, 2, 3, 4, 5])
print("1D Array:")
print("Array:", arr_1d)
print("Shape:", arr_1d.shape)  # Output: (5,)
print("Number of Dimensions:", arr_1d.ndim)  # Output: 1
print("Size:", arr_1d.size)  # Output: 5
print("Data Type:", arr_1d.dtype)  # Output: int64
print()

# Example 2: 2D Array. Here, we used nested lists to create a multi-dimensional array.
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:")
print("Array:\n", arr_2d)
print("Shape:", arr_2d.shape)  # Output: (2, 3)
print("Number of Dimensions:", arr_2d.ndim)  # Output: 2
print("Size:", arr_2d.size)  # Output: 6
print("Data Type:", arr_2d.dtype)  # Output: int64
print()
```

2. NumPy

2.5 ndarray Methods

Except for the previously introduced array creation methods, NumPy provides various methods for array manipulation, element-wise operations, broadcasting, indexing, slicing, aggregation, statistics, linear algebra, etc.

Array Manipulation:

- `ndarray.reshape()`: Change the shape of an array.
- `ndarray.flatten()`: Return a 1D copy of the array.
- `ndarray.transpose()`: Return the transpose of the array.
- `ndarray.swapaxes()`: Swap two axes of an array.
- `ndarray.split()`: Split an array into multiple sub-arrays.
- `ndarray.resize()`: Resize an array in-place.

It's important to note that the behavior of `np.resize()` can lead to unexpected results if you're not careful with the dimensions and the relationship between the original and target shapes. If you want to avoid data repetition or truncation, you should use `np.reshape()` or other array manipulation functions to ensure that the new shape is compatible with the original array's shape and size.

```python
import numpy as np

# Example Array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# ndarray.reshape(): Change the shape of an array
reshaped = arr.reshape((1, 9))
print("Reshaped Array:\n", reshaped)

# ndarray.flatten(): Return a 1D copy of the array
flattened = arr.flatten()
print("Flattened Array:", flattened)

# ndarray.transpose(): Return the transpose of the array
transposed = arr.transpose()
print("Transposed Array:\n", transposed)

# ndarray.swapaxes(): Swap two axes of an array
swapped_axes = arr.swapaxes(0, 1)
print("Swapped Axes Array:\n", swapped_axes)

# ndarray.split(): Split an array into multiple sub-arrays
split_arrays = np.split(arr, 3, axis=0)
print("Split Arrays:", split_arrays)

# np.resize(): Resize an array using the np.resize function
resized_array = np.resize(arr, (4, 3))
print("Resized Array:\n", resized_array)
```

2. NumPy

2.5 ndarray Methods

Element-wise Operations:

- Arithmetic operators: +, -, *, /, ** (power), etc.
- `numpy.add()`, `numpy.subtract()`, `numpy.multiply()`, `numpy.divide()`: Element-wise arithmetic functions.
- `numpy.exp()`, `numpy.log()`, `numpy.sin()`, `numpy.cos()`, `numpy.sqrt()`, etc.: Element-wise mathematical functions. (Compute the exponential/log/sin/cos/square root of each element)

```
import numpy as np

# Example Arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# Arithmetic operations: +, -, *, /, **
(power), etc.
addition = arr1 + arr2
subtraction = arr1 - arr2
multiplication = arr1 * arr2
division = arr1 / arr2
power = arr1 ** 2
```

```
Addition:
 [[ 6  8]
 [10 12]]
Subtraction:
 [[-4 -4]
 [-4 -4]]
Multiplication:
 [[ 5 12]
 [21 32]]
Division:
 [[0.2        0.33333333]
 [0.42857143 0.5       ]]
Power:
 [[ 1  4]
 [ 9 16]]
```

Check the supplied materials (4.Element-wiseOperations.txt) for more examples

2. NumPy

2.5 ndarray Methods

Array Broadcasting:

Broadcasting automatically adjusts smaller arrays' dimensions to match larger arrays' dimensions, making element-wise operations between arrays of different shapes possible. It follows two rules:

1.Dimension Prepending Rule: When input arrays have different dimensions, smaller arrays are prepended with "1"s to match the largest array's dimensions.

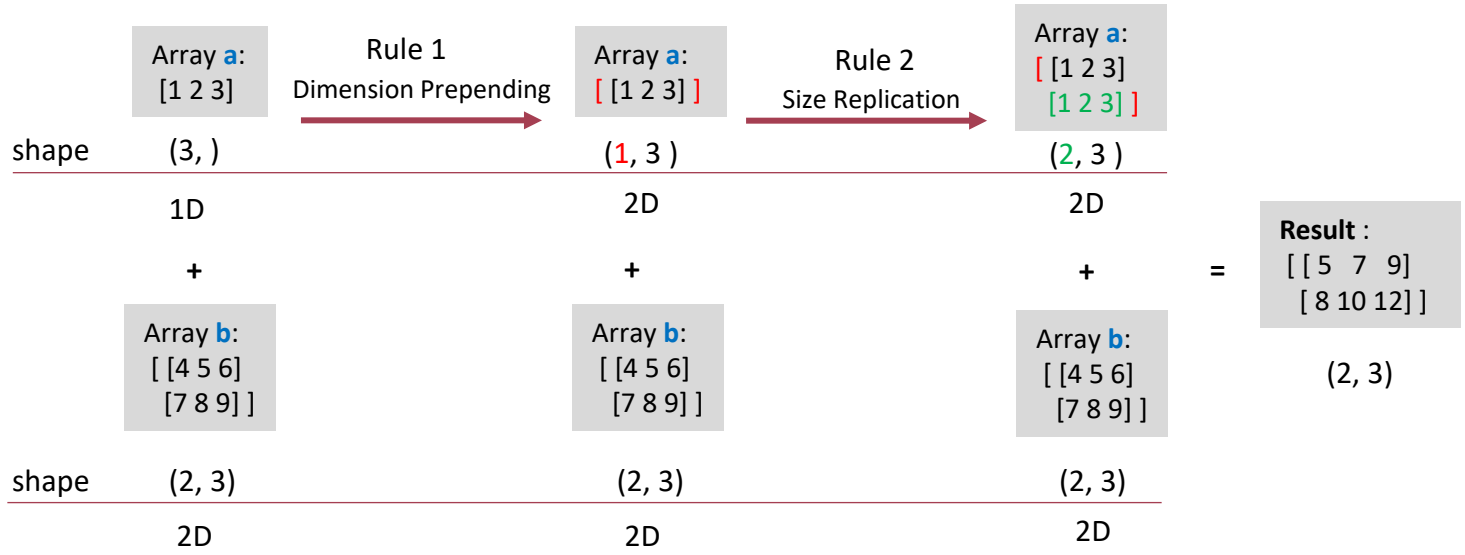
2.Size Replication Rule: Arrays with size 1 along a dimension are treated as if they have the size of the largest array along that dimension. The values in the smaller array are replicated along that dimension for broadcasting.

```
import numpy as np

# Example arrays
a = np.array([1, 2, 3])    # Shape: (3,)
b = np.array([[4, 5, 6],   # Shape: (2, 3)
              [7, 8, 9]])

# Broadcasting and element-wise addition
result = a + b

print("Array a:\n", a)
print("Array b:\n", b)
print("Result after broadcasting and addition:\n", result)
```

Array a:
[1 2 3]

Array b:
[[4 5 6]
[7 8 9]]

Result after broadcasting and addition:
[[5 7 9]
[8 10 12]]



2. NumPy

2.5 ndarray Methods

Array Broadcasting Application:

One important aspect of broadcasting is calculating functions on regularly spaced grids. For example, creating the multiplication table:

```
import numpy as np

# Get a 1D array contains numbers 1 to 9
a = np.arange(1, 10)
print("Array a:", a)
print("Array a's shape:", a.shape)

# In NumPy, X[:, np.newaxis] returns an array with
# additional 1's appended to the shape of array X
a_reshaped = a[:, np.newaxis]
print()
print("Array a_reshaped:\n")
print(a_reshaped)
print("Array a_reshaped's shape:", a_reshaped.shape)

# Calculate the multiplication table
table = a_reshaped * a
print()
print("Multiplication table:\n")
print(table)
```

```
Array a: [1 2 3 4 5 6 7 8 9]
Array a's shape: (9,)

Array a_reshaped:

[[1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]
 [9]]
Array a_reshaped's shape: (9, 1)
```

```
Multiplication table:

[[ 1  2  3  4  5  6  7  8  9]
 [ 2  4  6  8 10 12 14 16 18]
 [ 3  6  9 12 15 18 21 24 27]
 [ 4  8 12 16 20 24 28 32 36]
 [ 5 10 15 20 25 30 35 40 45]
 [ 6 12 18 24 30 36 42 48 54]
 [ 7 14 21 28 35 42 49 56 63]
 [ 8 16 24 32 40 48 56 64 72]
 [ 9 18 27 36 45 54 63 72 81]]
```

2. NumPy

2.5 ndarray Methods

Array Broadcasting and Element-wise Comparison:

- `numpy.broadcast_to()`: Broadcast an array to a specified shape.
- Comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`, etc. (Element-wise comparison)
- `ndarray.all()`: Check if all elements are True.
- `ndarray.any()`: Check if any element is True.
- `numpy.logical_and()`, `numpy.logical_or()`, `numpy.logical_not()`: Element-wise logical operations.

```python
import numpy as np

# Example Arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[1, 2], [3, 3]])

# numpy.broadcast_to(): Broadcast an array to a specified shape
broadcasted = np.broadcast_to(arr1, (2, 2, 2))
print("Broadcasted Array:\n", broadcasted)

# Comparison operators: ==, !=, <, <=, >, >=, etc.
comparison_equal = arr1 == arr2
comparison_not_equal = arr1 != arr2
comparison_less_than = arr1 < arr2
comparison_less_than_equal = arr1 <= arr2
comparison_greater_than = arr1 > arr2
comparison_greater_than_equal = arr1 >= arr2
print("Comparison (Equal):\n", comparison_equal)
print("Comparison (Not Equal):\n", comparison_not_equal)
print("Comparison (Less Than):\n", comparison_less_than)
print("Comparison (Less Than or Equal):\n", comparison_less_than_equal)
print("Comparison (Greater Than):\n", comparison_greater_than)
print("Comparison (Greater Than or Equal):\n", comparison_greater_than_equal)
```

```
Broadcasted Array:
 [[[1 2]
   [3 4]]

  [[1 2]
   [3 4]]]
```

```python
# ndarray.all(): Check if all elements are True
all_true = arr1.all()
print("All elements are True:", all_true)

# ndarray.any(): Check if any element is True
any_true = arr1.any()
print("Any element is True:", any_true)

# numpy.logical_and(), numpy.logical_or(), numpy.logical_not(): Element-
# wise logical operations
logical_and = np.logical_and(arr1 > 1, arr2 > 2)
logical_or = np.logical_or(arr1 > 2, arr2 > 2)
logical_not = np.logical_not(arr1 > 2)

print("Logical AND:\n", logical_and)
print("Logical OR:\n", logical_or)
print("Logical NOT:\n", logical_not)
```

```
Comparison (Less Than or Equal):
 [[ True  True]
  [ True False]]
```

2. NumPy

2.5 ndarray Methods

Array Indexing and Slicing:

- Basic indexing and slicing using square brackets "[ ]".
- Fancy indexing with integer or boolean arrays.
- ndarray[condition]: Select elements that satisfy a condition.

```python
import numpy as np

# Create a 3D array
arr_3d = np.array([[[ 1,  2,  3],
                    [ 4,  5,  6]],

                   [[ 7,  8,  9],
                    [10, 11, 12]]])

# Display the original 3D array
print("Original 3D Array:\n", arr_3d)

# Slicing along different dimensions
print("\nSlicing Examples:")
# Output: First plane of the 3D array
print("arr_3d[0, :, :]:\n", arr_3d[0, :, :])
# Output: Second row of each plane
print("arr_3d[:, 1, :]:\n", arr_3d[:, 1, :])
# Output: Third column of each plane
print("arr_3d[:, :, 2]:\n", arr_3d[:, :, 2])
```

```
Original 3D Array:
 [[[ 1  2  3]
   [ 4  5  6]]

  [[ 7  8  9]
   [10 11 12]]]

Slicing Examples:
arr_3d[0, :, :]:
 [[1 2 3]
  [4 5 6]]
arr_3d[:, 1, :]:
 [[ 4  5  6]
  [10 11 12]]
arr_3d[:, :, 2]:
 [[ 3  6]
  [ 9 12]]
```

Multidimensional array slicing example

```python
import numpy as np

# Example array
arr = np.array([10, 20, 30, 40, 50])

# Basic indexing and slicing
print("Basic Indexing and Slicing:")
# Access element at index 0
print("arr[0]:", arr[0])         # Output: 10
# Slice from index 1 to 3 (exclusive)
print("arr[1:4]:", arr[1:4])     # Output: [20 30 40]
# Fancy indexing with integer arrays
print("\nFancy Indexing (Integer):")
# Access elements at indices 1 and 3
indices = np.array([1, 3])
print("arr[indices]:", arr[indices])  # Output: [20 40]

# Fancy indexing with boolean array
print("\nFancy Indexing (Boolean):")
# Access elements with corresponding True values in mask
mask = np.array([True, False, True, False, True])
print("arr[mask]:", arr[mask])        # Output: [10 30 50]

# Indexing with condition
print("\nIndexing with Condition:")
# Access elements that satisfy the condition (greater than 25)
condition = arr > 25
print("arr[condition]:", arr[condition])  # Output: [30 40 50]
```

1D array indexing and slicing example

2. NumPy

2.5 ndarray Methods

Array Aggregation and Statistics:

- `ndarray.sum(), ndarray.mean(), ndarray.std(), ndarray.var():` Compute various statistics.
- `ndarray.min(), ndarray.max(), ndarray.argmin(), ndarray.argmax()`: Find minimum, maximum, and their indices.
- `ndarray.median(), ndarray.percentile(), ndarray.histogram():` Calculate other statistical measures.

Array Linear Algebra:

- `numpy.dot(), numpy.matmul():` Perform matrix multiplication.
- `numpy.linalg.inv(), numpy.linalg.det():` Compute matrix inverse and determinant.
- `numpy.linalg.eig(), numpy.linalg.svd():` Compute eigenvalues and singular value decomposition.

Array Reshaping and Resizing:

- `ndarray.reshape(), ndarray.resize():` Change the shape of an array.
- `numpy.ravel(), ndarray.flatten():` Convert multi-dimensional arrays to 1D

Array Sorting:

- `ndarray.sort():` Sort elements along a specified axis.

Array Copying and Views:

- `ndarray.copy():` Create a deep copy of the array.
- `ndarray.view():` Create a new view of the array with the same data.

Array I/O:

- `numpy.save(), numpy.savez():` Save arrays to binary files.
- `numpy.load():` Load arrays from binary files.

Check the documentation for more methods.

2. NumPy

2.6 Iteration Over Arrays

- NumPy element-wise operations are two orders of magnitude faster than explicit looping. You should first consider employing these existing operations to address your algorithms.
- If explicit looping is needed, using Cython or Numba compiles the loop into the machine code can archive comparable performance of the existing operations.
- NumPy supports both the general Python iteration and a more flexible `nditer` iterator.
  - Simply looping over an `ndarray` returns a sequence of elements that varies along the first dimensions.
  - The `nditer` iterator provides many flexible ways to visit all the elements of one or more arrays.

```python
import numpy as np

# Create a 2D array
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

# Iterate over the array using nested loops
print("Iterating over the array:")
for row in arr:
    for element in row:
        print(element, end=' ')
    print()  # Move to the next row

# Alternatively, you can use nditer to
# iterate over the array efficiently
print("\nIterating using nditer:")
for element in np.nditer(arr):
    print(element, end=' ')
```

```
Iterating over the array:
1 2 3
4 5 6
7 8 9

Iterating using nditer:
1 2 3 4 5 6 7 8 9
```

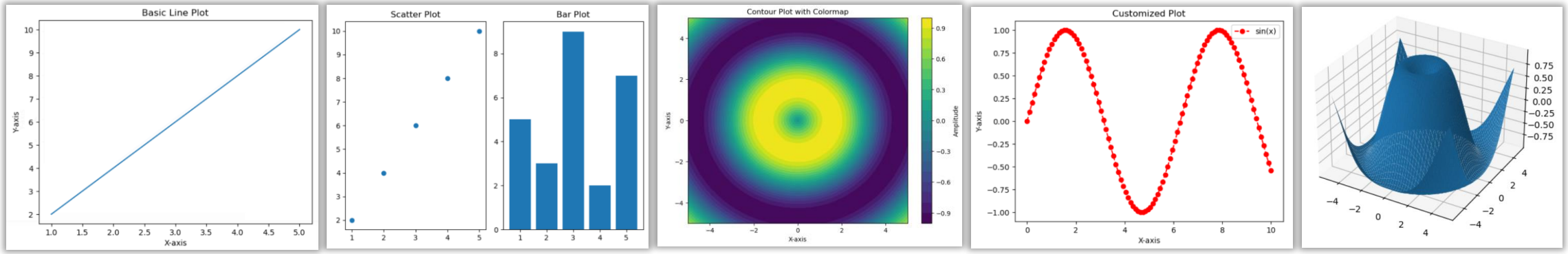
There are more options with the `nditer` iterator, such as controlling the order of iteration and iterating with broadcasting. Check the documentation for details.

3. Matplotlib

Python’s Data Visualization Powerhouse

3. Matplotlib

- Matplotlib is a widely used Python library for creating static, animated, and interactive visualizations in various formats. Anaconda has Matplotlib installed by default.
- It provides a flexible and powerful framework for generating high-quality plots, charts, and other visual representations of data.
- Features:
  - **Basic Plotting**: such as line plots, scatter plots, bar plots, histograms, and more
  - **Multiple Subplots**: create multiple subplots within a single figure
  - **Colormaps and Colorbars**: help represent data values using colors
  - **3D Plotting**: create three-dimensional plots for visualizing data in 3D
  - **Customization, Text, and Annotations**: customize many aspects of your plots, such as colors, line styles, markers, labels, titles, axes, grid lines, and legends. Add text, annotations, and arrows to your plots for additional context and explanations.
  - **Interactive Features:** interactive environments like Jupyter Notebooks, allow you to create plots that respond to user inputs
  - **Export and Saving**: export plots to various file formats, including PNG, JPEG, PDF, SVG, and more.
  - **Seamless Integration**: can be used alongside other libraries like NumPy, Pandas, and SciPy for comprehensive data analysis and visualization



3. Matplotlib

3.1 Matplotlib Architecture

- **Scripting Layer:** it provides the pyplot interface (module) to users for creating and modifying plots.
- **Artist Layer:** defines the basic building blocks of a plot, like figures, axes, lines, text, rectangles, circles, and images.
- **Backend Layer**: responsible for rendering and displaying the graphics created using Matplotlib. Its event-handling system allows users to respond to user interactions with the plot.

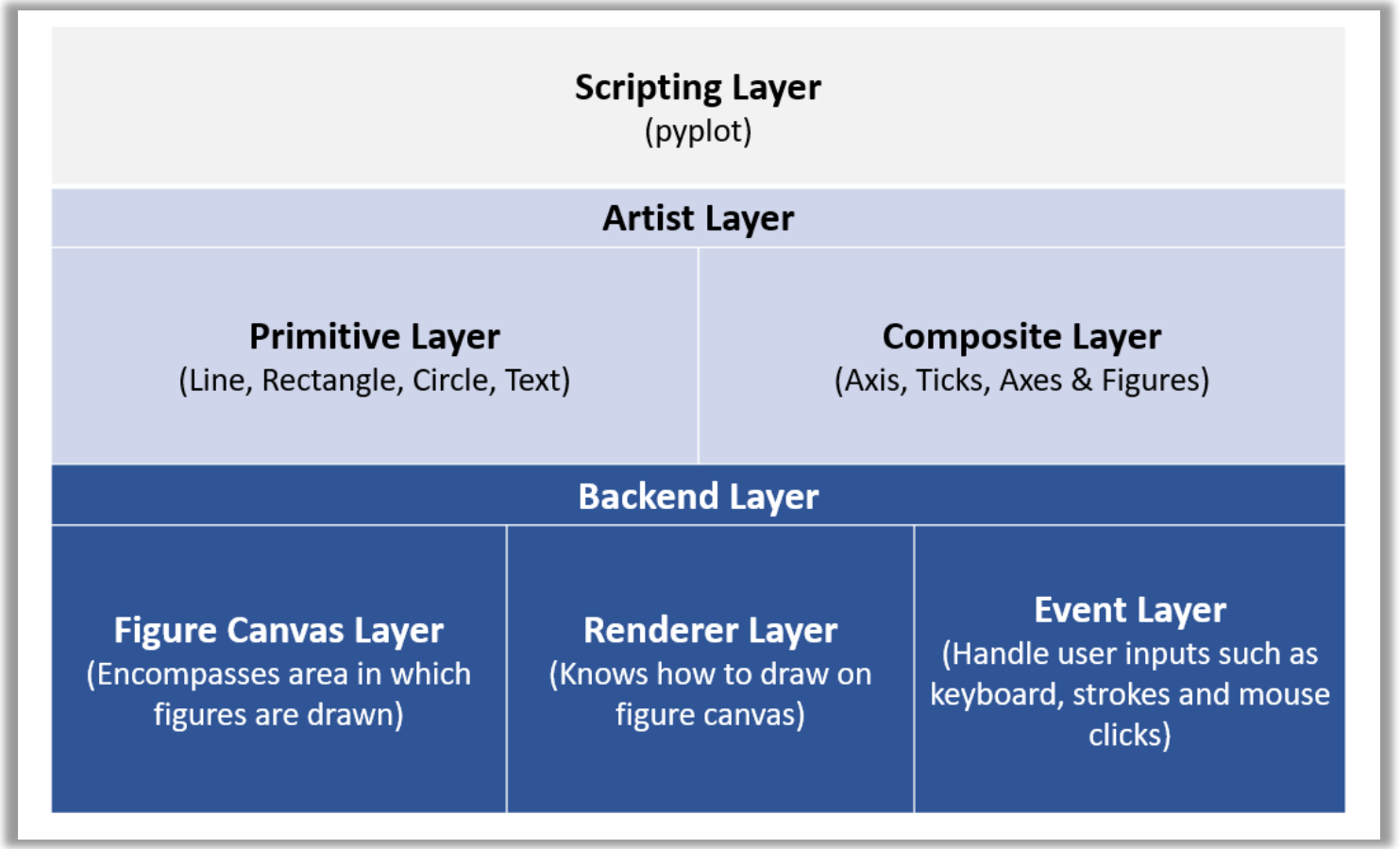
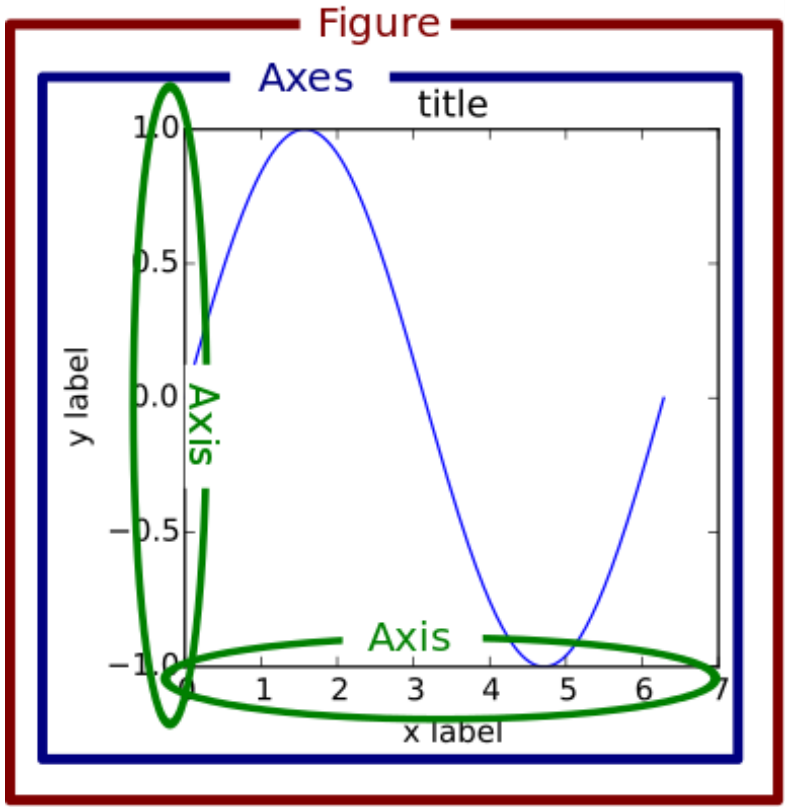


Figure source: https://www.kaggle.com/discussions/getting-started/146720

3. Matplotlib

3.2 Matplotlib Object Hierarchy

The **figure** is a top-level container that may contain one or multiple **axes** (which are plots)



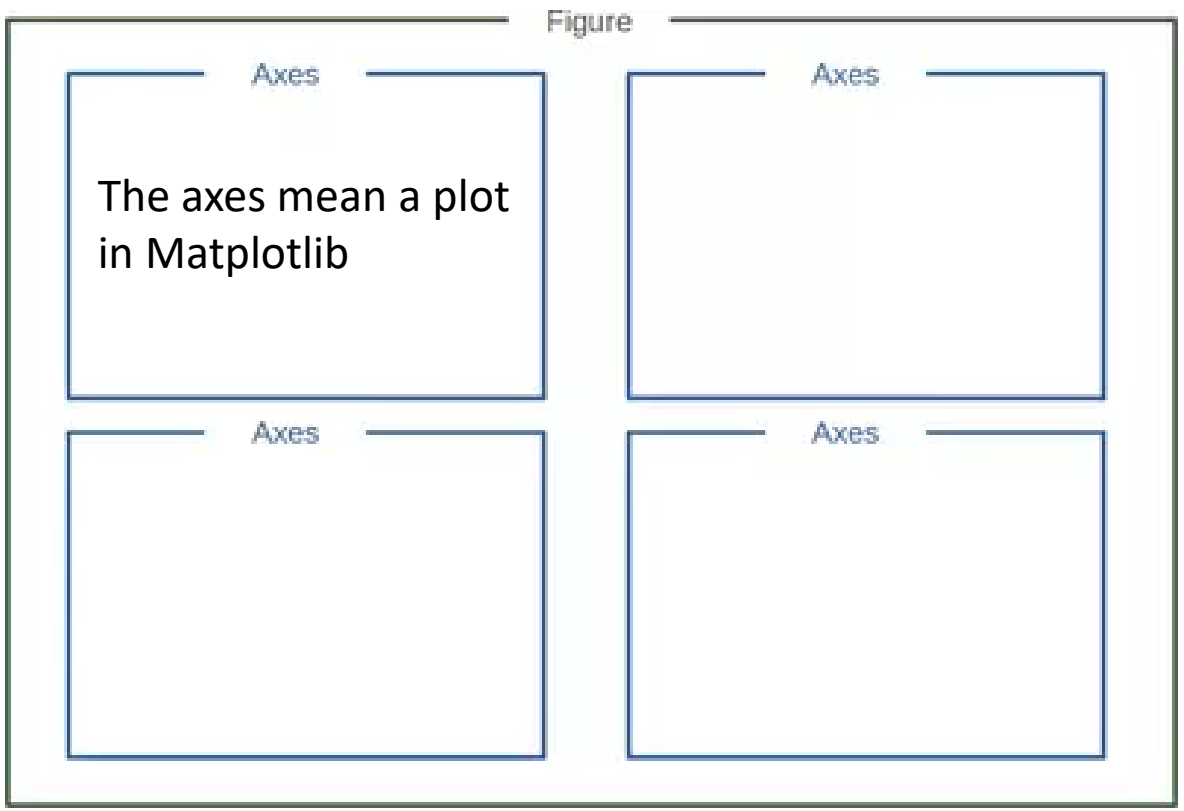
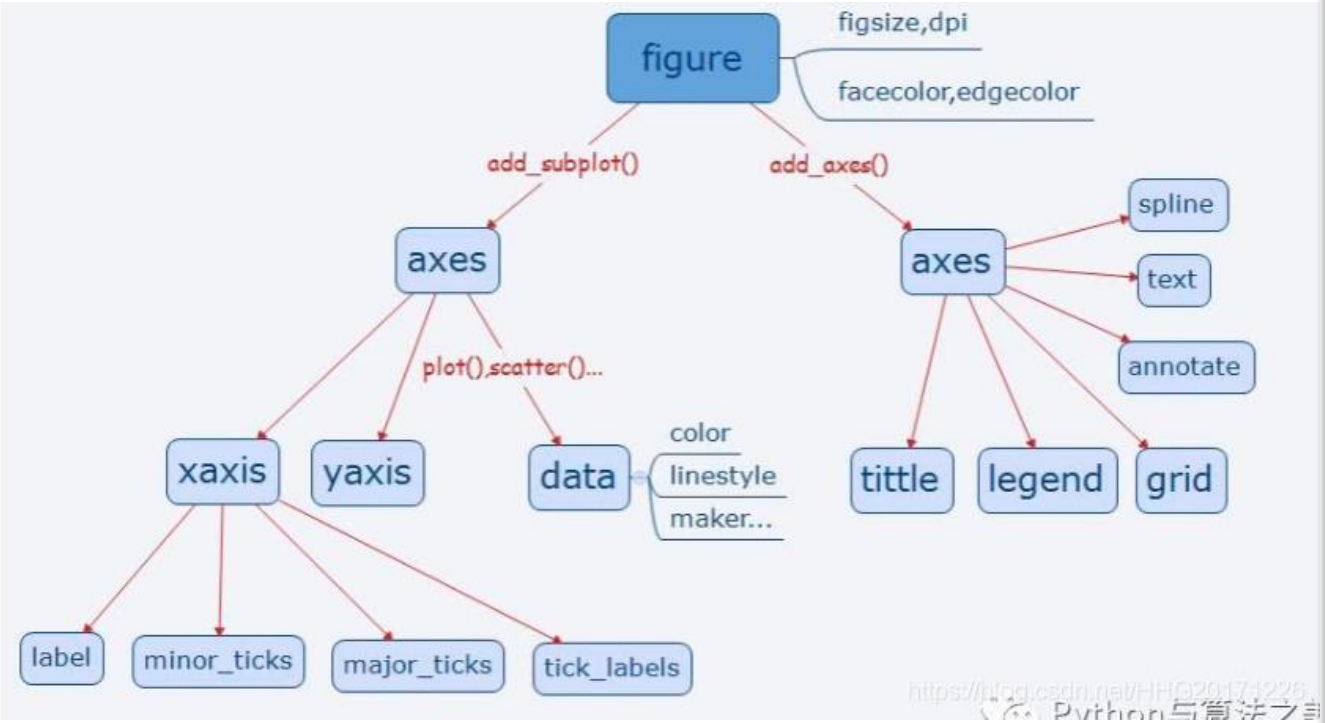


Figure source: https://python-course.eu/numerical-programming/matplotlib-object-hierarchy.php
https://realpython.com/python-matplotlib-guide/

3. Matplotlib

3.2 Matplotlib Object Hierarchy

Except for the **axes** objects, The **figure** object has a few properties, such as a title, size, and background/border colour. The **axes** object has its own components shown here.



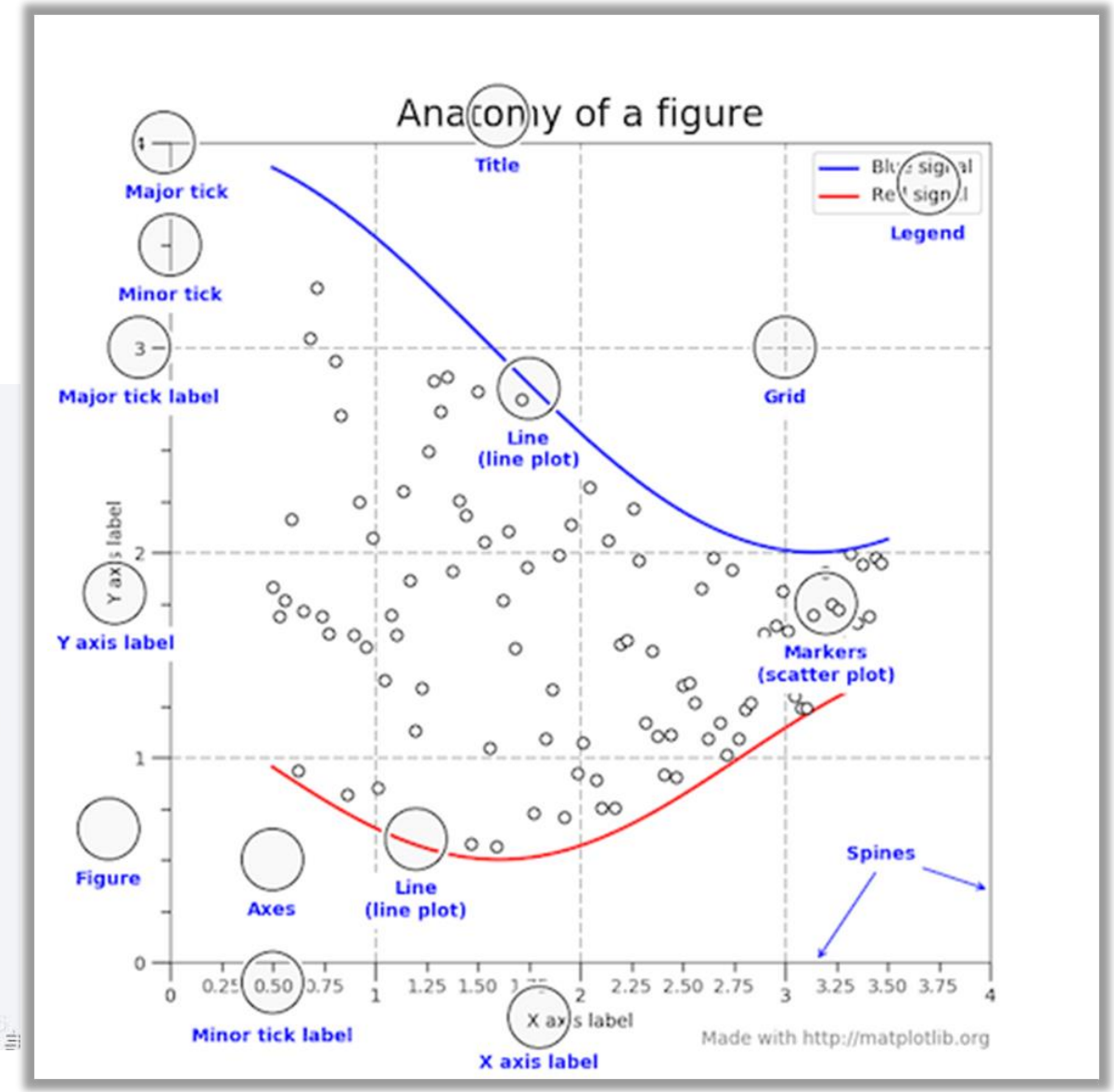
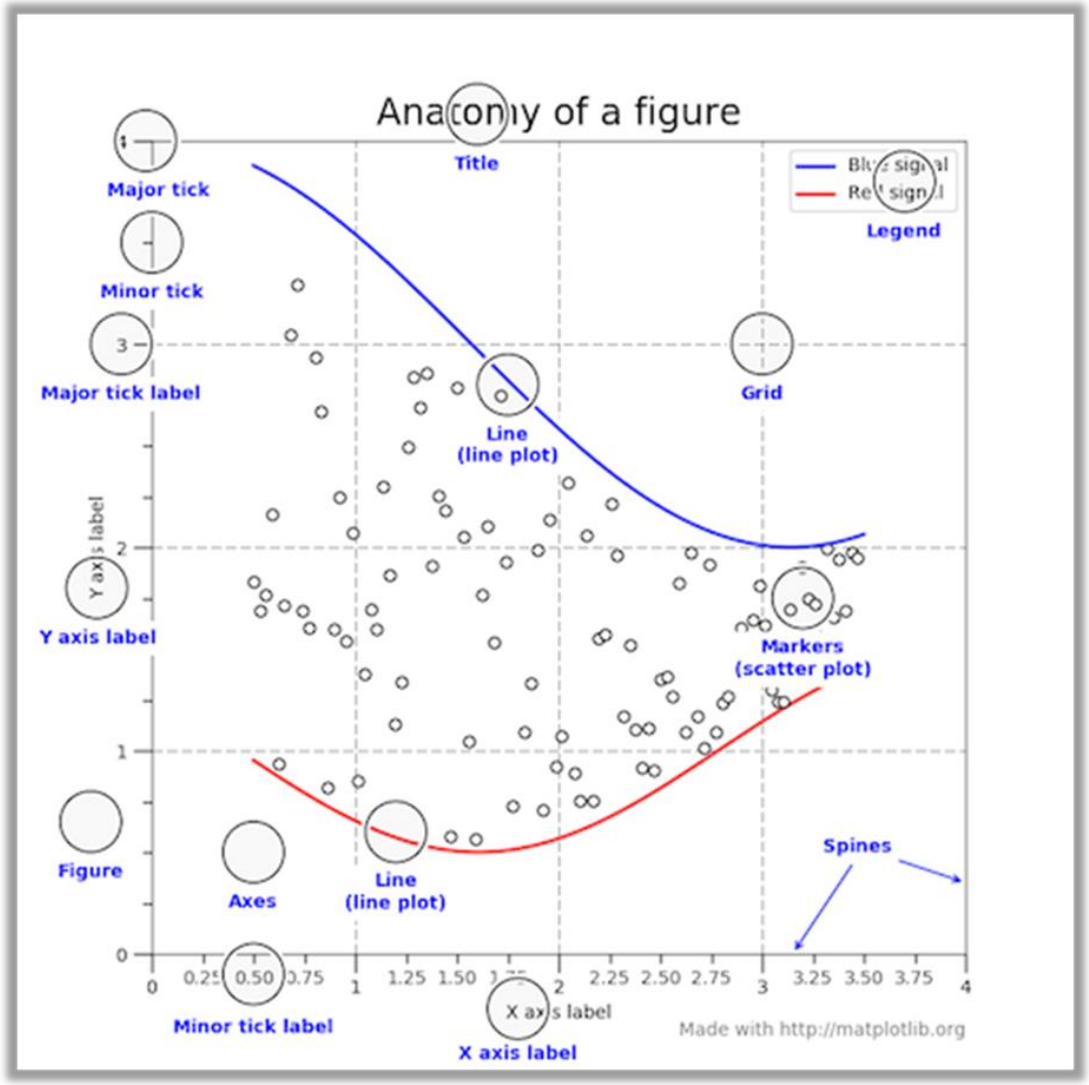


Figure source: https://blog.csdn.net/HHG20171226/article/details/103192399

3. Matplotlib

3.3 Matplotlib Figure Elements



1. **Figure:** The top-level container that encompasses the entire plot
2. **Axes**: A region within a figure where data is plotted.
3. **Axis**: The x-axis or y-axis of an Axes
4. **Lines**: Represents a line plot
5. **Markers:** Markers represent individual data points on the plot
6. **Text**: Displays text on the plot, including titles, labels, and annotations
7. **Legends**: Displays a legend that explains the mapping between data and plot elements
8. **Annotations:** Provides textual annotations on the plot
9. **Spines**: The edges of the plot's frame that demarcate the data region
10. **Ticks:** Marks and labels along the axes to indicate data values.

3. Matplotlib

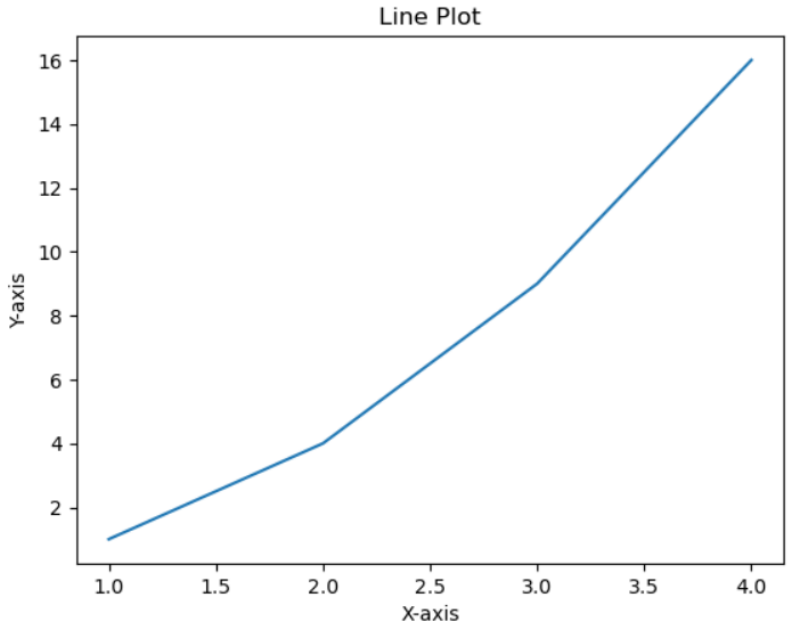
3.4 Interfaces for Creating Plots

Stateful (State-Based) Interface:
Stateful interface is an easier way to make plots in Matplotlib. It follows a procedural approach where you create plots using a series of function calls. This interface is suitable for quick and interactive plotting tasks.

Key Features:

- Implicitly maintains an active figure and axes in the background.
- Relies heavily on the plt module (Matplotlib's pyplot module).
- Functions like plt.plot(), plt.scatter(), plt.title(), etc., directly affect the active plot.
- Convenient for quick exploratory data visualization.

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4]
y = [1, 4, 9, 16]

# Create a line plot using stateful interface
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')
plt.show()
```



3. Matplotlib

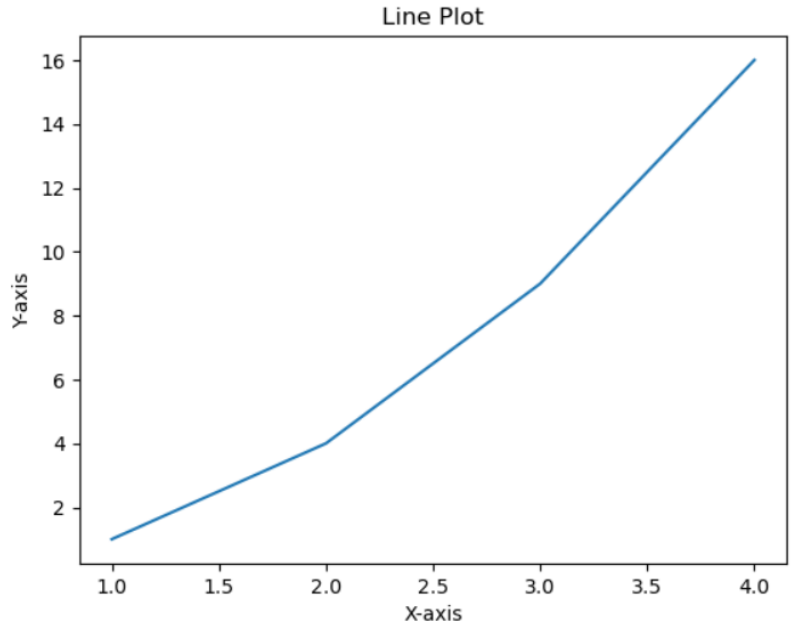
3.4 Interfaces for Creating Plots

Stateless (Object-Oriented) Interface:
The stateless interface is also called the object-oriented interface and gives you more plot control. It involves creating explicit figures and axes objects and manipulating them directly. This approach is more flexible and suited for complex or reusable plots.

Key Features:

- Involves creating Figure and Axes objects explicitly.
- Provides greater control over individual plot elements and their properties.
- Encourages a more organized and modular code structure.
- Suitable for creating multiple subplots or advanced visualizations.
- We recommend it for more structured and production-level code.

```python
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4]
y = [1, 4, 9, 16]

# Create Figure and Axes objects using OO interface
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Line Plot')
plt.show()
```



3. Matplotlib

3.5 `plt.subplots()` Function

It is used to create a grid of subplots within a single figure.

Signature: `plt.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot_kw=None, gridspec_kw=None, **fig_kw)`

- **nrows** and **ncols**: These parameters specify the number of rows and columns in the grid of subplots. They determine the layout of the subplots within the figure.
- **sharex** and **sharey**: These optional parameters determine whether the x-axis (sharex) or y-axis (sharey) will be shared among subplots. This can be useful for comparing different plots with the same axis scales.
- **squeeze**: When squeeze is True (default), if only one subplot is created, the returned axes will be a single Axes object rather than a 2D NumPy array.
- **subplot_kw**: This is a dictionary of keyword arguments that will be passed to the add_subplot() method when creating each individual subplot.
- **gridspec_kw**: This is a dictionary of keyword arguments that will be passed to the GridSpec instance used to create the grid layout of subplots.
- ****fig_kw**: Any additional keyword arguments are passed to the `plt.figure()` function, allowing you to customize the figure itself, such as its size, background color, etc.

3. Matplotlib

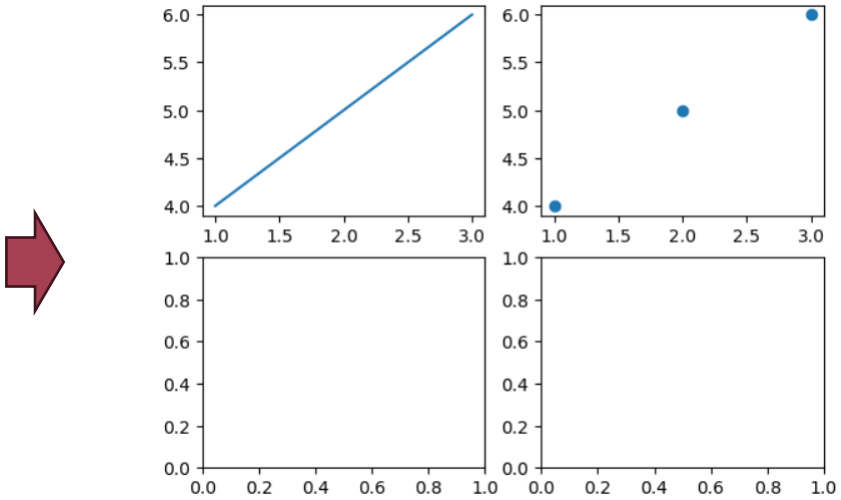
3.5 `plt.subplots()` Function

It is used to create a grid of subplots within a single figure.

Return values: `fig, axes = plt.subplots(nrows, ncols, ...optional_parameters...)`

- **fig**: This is a Figure object, which serves as the top-level container for all the subplots and other plot elements. You can use this Figure object to control the properties of the entire figure, such as its size and title.
- **axes**: This is a NumPy array containing Axes objects. Each Axes object represents an individual subplot within the figure. You can use these Axes objects to customize and plot data for each subplot. With the default parameters, it will be a single Axes object rather than a 2D NumPy array.
  - The axes array has a shape determined by the `nrows` and `ncols` parameters. The size of the array will be `(nrows, ncols)`.

```
import matplotlib.pyplot as plt

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=2)

# Customize and plot on individual subplots
axes[0, 0].plot([1, 2, 3], [4, 5, 6])
axes[0, 1].scatter([1, 2, 3], [4, 5, 6])

# Display the figure with subplots
plt.show()
```

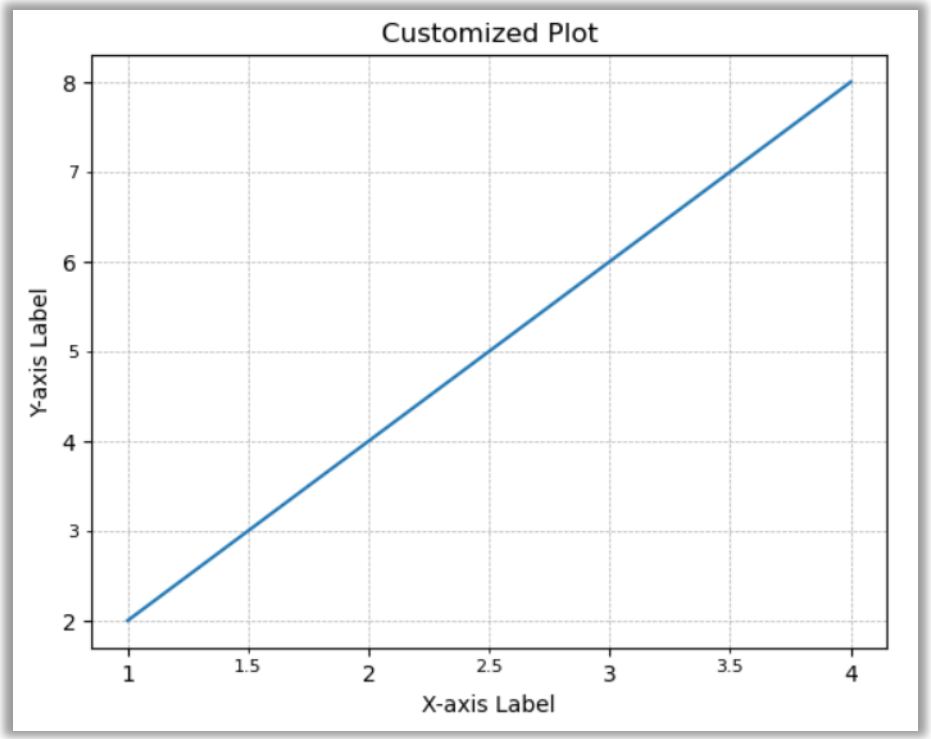


3. Matplotlib

3.6 Example 1: Adding plot title, major/minor ticks and related labels, and a grid

```
import matplotlib.pyplot as plt

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Plot some data (you can replace this with your own
data)
x = [1, 2, 3, 4]
y = [2, 4, 6, 8]
ax.plot(x, y)

# Set axes title
ax.set_title("Customized Plot")

# Set x and y axis labels
ax.set_xlabel("X-axis Label")
ax.set_ylabel("Y-axis Label")

# Set major and minor ticks on x and y axes
ax.set_xticks([1, 2, 3, 4])
ax.set_yticks([2, 4, 6, 8])
ax.set_xticks([1.5, 2.5, 3.5], minor=True)
ax.set_yticks([3, 5, 7], minor=True)

# Set labels for major and minor ticks
ax.set_xticklabels(["1", "2", "3", "4"])
ax.set_yticklabels(["2", "4", "6", "8"])
ax.set_xticklabels(["1.5", "2.5", "3.5"], minor=True)
ax.set_yticklabels(["3", "5", "7"], minor=True)
```

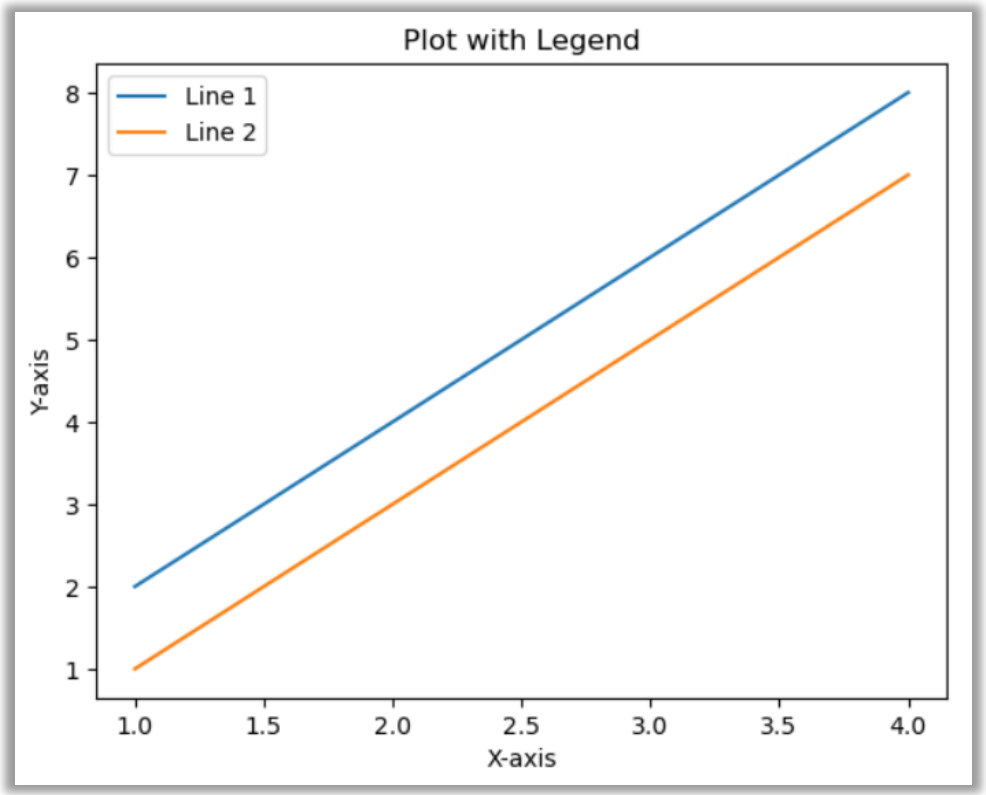
```
# Set font size for minor tick labels
ax.tick_params(axis='both', which='minor', labelsize=8)

# Turn on the grid
ax.grid(True, which='both', linestyle='--', linewidth=0.5)

# Display the plot
plt.show()
```



3. Matplotlib

3.6 Example 2: Adding a legend

```
import matplotlib.pyplot as plt

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Plot some data
x = [1, 2, 3, 4]
y1 = [2, 4, 6, 8]
y2 = [1, 3, 5, 7]
line1, = ax.plot(x, y1, label='Line 1')
line2, = ax.plot(x, y2, label='Line 2')

# Set axes title
ax.set_title("Plot with Legend")

# Set x and y axis labels
ax.set_xlabel("X-axis")
ax.set_ylabel("Y-axis")

# Add a legend
ax.legend(loc='upper left')

# Display the plot
plt.show()
```

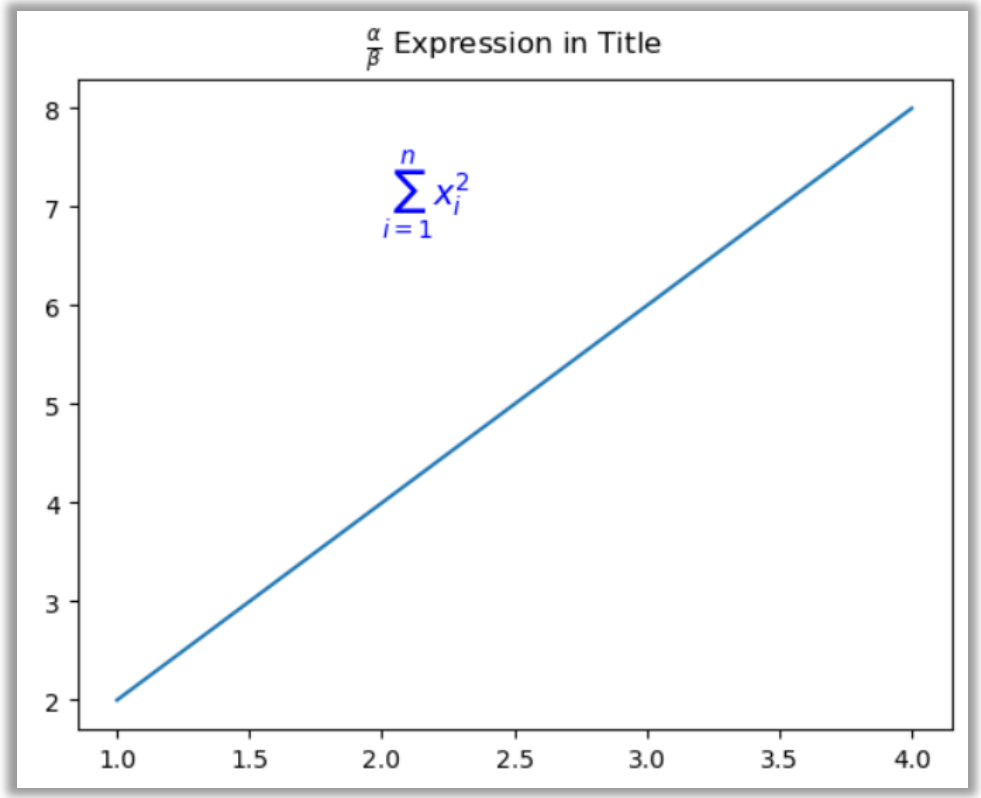


3. Matplotlib

3.6 Example 3: Adding a LaTeX expression to text

- LaTeX expressions are used to render mathematical symbols and equations. We can insert them beautifully within the plot title and text.
- We can add a LaTeX expression to a text by prefixing the string with `r` (raw string) and enclosing the expression in dollar signs (`$`).

```
import matplotlib.pyplot as plt

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Set axes title with a LaTeX expression
ax.set_title(r"$\frac{\alpha}{\beta}$ Expression in Title")

# Plot some data
x = [1, 2, 3, 4]
y = [2, 4, 6, 8]
ax.plot(x, y)

# Add text with a LaTeX expression
ax.text(2, 7, r"$\sum_{i=1}^{n} x_i^2$", fontsize=14, color='blue’)

# Display the plot
plt.show()
```



3. Matplotlib

3.6 Example 4: Saving a figure

- You can use the `savefig()` function to save the generated figure to a file at the end of the plotting commands.
- You can save the figure with different formats using the corresponding file extension names, such as .png, .jpg, .svg, pdf, etc.

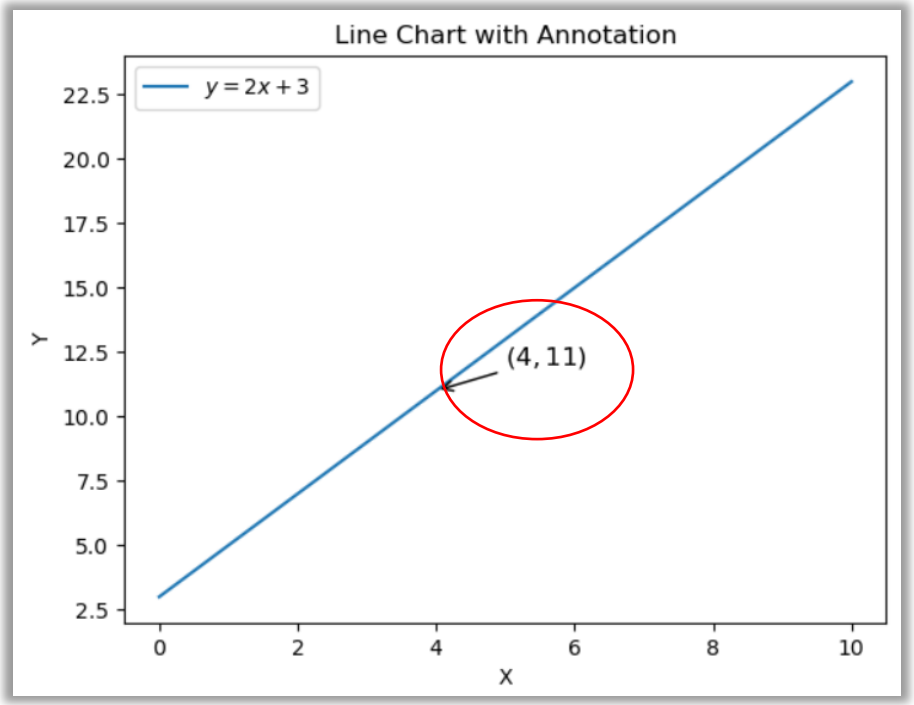
```
import matplotlib.pyplot as plt
import os

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Set axes title with a LaTeX expression
ax.set_title(r"$\frac{\alpha}{\beta}$ Expression in Title")

# Plot some data
x = [1, 2, 3, 4]
y = [2, 4, 6, 8]
ax.plot(x, y)

# Add text with a LaTeX expression
ax.text(2, 7, r"$\sum_{i=1}^{n} x_i^2$", fontsize=14, color='blue’)

# Save the figure to a file (PDF format in this example)
plt.savefig("my_plot.pdf")
print("Output figure location: ", os.getcwd())

# Display the plot
plt.show()
```

The `savefig()` function will save the generated figure in the current working directory, which can be acquired by using the `os.getcwd()` function from the standard library

3. Matplotlib

3.6 Example 5: Adding annotations

- You can use the `ax.annotate()` method to add an annotation to a specific point on the plot.
- The `xy` parameter specifies the point to annotate, and `xytext` specifies the text position. You can also specify the arrow style with the `arrowprops` parameter.

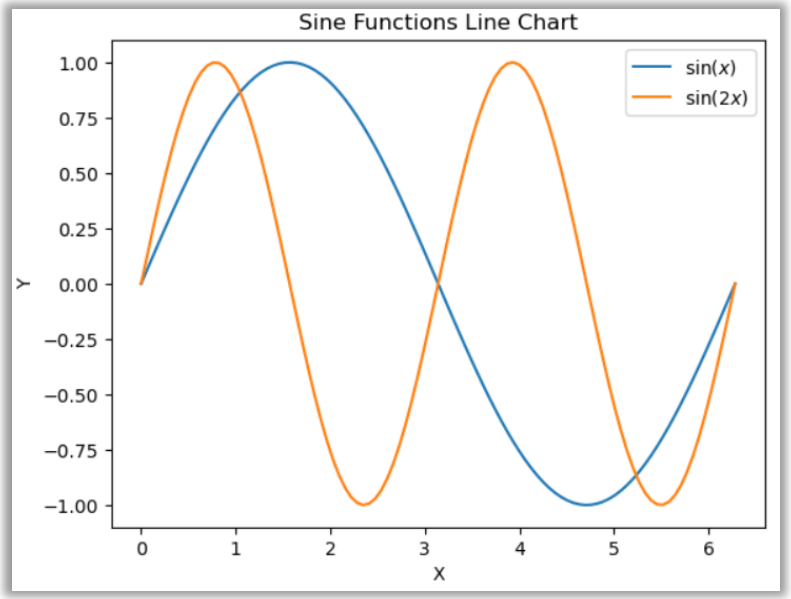



```python
import numpy as np
import matplotlib.pyplot as plt

# Generate data
x = np.linspace(0, 10, 100)
y = 2 * x + 3

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Plot the line chart
ax.plot(x, y, label=r"$y = 2x + 3$")

# Annotate a point on the line with a LaTeX expression
annotate_x = 4
annotate_y = 2 * annotate_x + 3
ax.annotate(r"$(4, 11)$", xy=(annotate_x, annotate_y), xytext=(5, 12),
            arrowprops=dict(arrowstyle="->"), fontsize=12)

# Set axes labels and title
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_title("Line Chart with Annotation")

# Add legend
ax.legend()

# Display the plot
plt.show()
```

3. Matplotlib

3.6 Example 6: Line chart

- In a line chart, straight lines connect individual data points, creating a visual representation of the overall trend or pattern.
- The x-axis represents the independent variable (such as time, category, or value), while the y-axis represents the dependent variable.




```python
import numpy as np
import matplotlib.pyplot as plt

# Generate x values
x = np.linspace(0, 2 * np.pi, 100)

# Calculate sine values for two functions
y1 = np.sin(x)
y2 = np.sin(2 * x)

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Plot the two sine functions
ax.plot(x, y1, label=r"$\sin(x)$")
ax.plot(x, y2, label=r"$\sin(2x)$")

# Set axes labels and title
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_title("Sine Functions Line Chart")

# Add legend
ax.legend()

# Display the plot
plt.show()
```

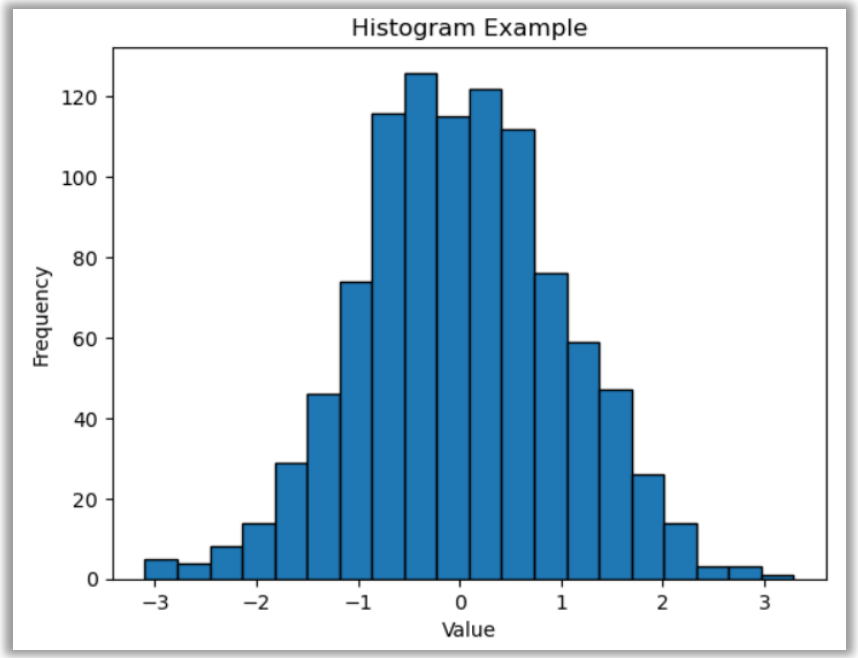
3. Matplotlib

3.6 Example 7: Histogram

- A histogram is a graphical representation of the distribution of a dataset. It provides a visual summary of the frequency or count of data values falling into specified intervals, called bins. Each bin represents a specific range of values.
- **Frequency or Count**: The vertical axis represents the frequency or count of data points in each bin.
- **X-Axis**: The x-axis represents the range of data values or the midpoints of each bin.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate random data
data = np.random.randn(1000)

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Create a histogram
ax.hist(data, bins=20, edgecolor='black')

# Set labels and title
ax.set_xlabel('Value')
ax.set_ylabel('Frequency')
ax.set_title('Histogram Example')

# Display the plot
plt.show()
```

The `bins` parameter specifies the number of bins or intervals for the histogram.



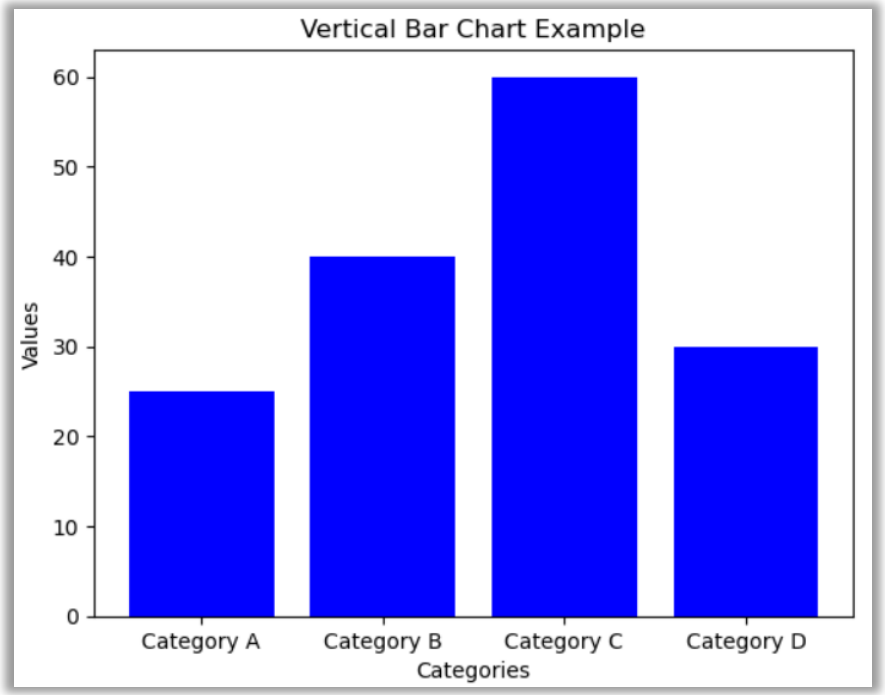
3. Matplotlib

3.6 Example 8: Bar chart

- A bar chart is similar to a histogram, however, the x-axis references categories instead of numerical values in this case.
- **Categories:** The x-axis represents the categories or labels of the data being plotted. Each category has its own bar.
- **Bar Length or Height:** Each bar's length (vertical bar chart) or height (horizontal bar chart) corresponds to the value associated with the category. The longer the bar, the larger the value.

```
import matplotlib.pyplot as plt

# Sample data
categories = ['Category A', 'Category B', 'Category C', 'Category D']
values = [25, 40, 60, 30]

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Create a vertical bar chart
ax.bar(categories, values, color='blue')

# Set labels and title
ax.set_xlabel('Categories')
ax.set_ylabel('Values')
ax.set_title('Vertical Bar Chart Example')

# Display the plot
plt.show()
```

You can create a horizontal bar chart by calling the `ax.barh()` method

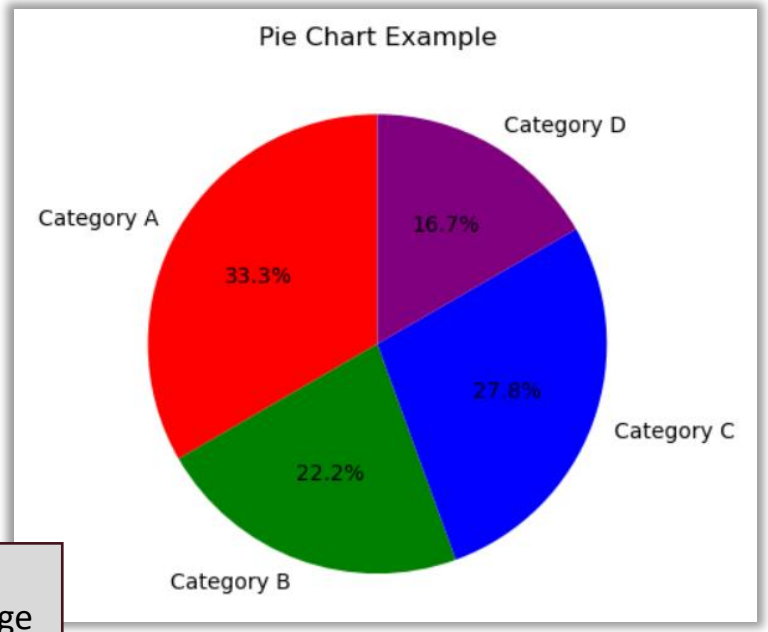


3. Matplotlib

3.6 Example 9: Pie chart

- Pie charts use slice sizes to show the relative category sizes.
- **Slices:** A slice of the pie represents each category. The size of each slice corresponds to the proportion of the total value it represents.
- **Color Coding:** Different colours are used to differentiate between slices. It's important to choose colours that are easily distinguishable.

```
import matplotlib.pyplot as plt

# Sample data
categories = ['Category A', 'Category B', 'Category C', 'Category D']
values = [30, 20, 25, 15]

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Create a pie chart
ax.pie(values, labels=categories, autopct='%1.1f%%', startangle=90,
colors=['red', 'green', 'blue', 'purple'])

# Set title
ax.set_title('Pie Chart Example')

# Display the plot
plt.show()
```

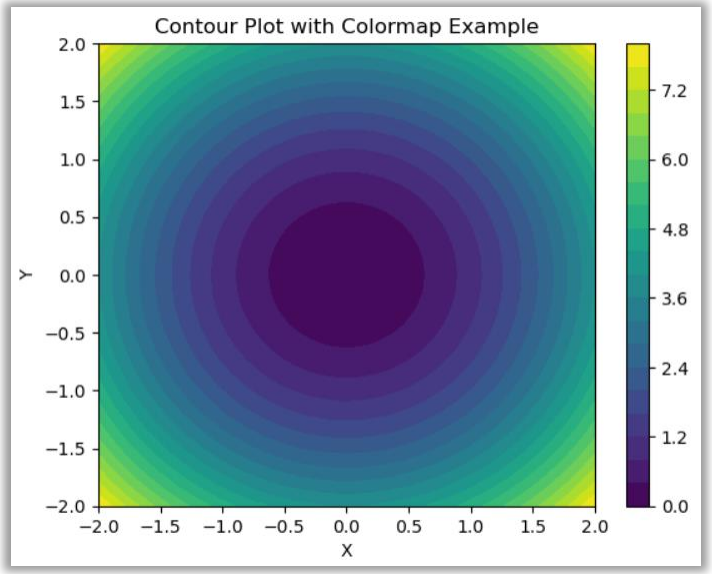


The `labels` parameter specifies the labels for each slice, the `autopct` parameter formats the percentage labels on the slices, and the `startangle` parameter sets the initial angle for the first slice.

3. Matplotlib

3.6 Example 10: Colormap

- A colormap (color map) maps numerical values to colors.
- **Color Range/Level:** A colormap defines a range of colors that correspond to different values within a data range.
- **Colormap Types:** Common types of colormaps include sequential colormaps (e.g., viridis), diverging colormaps (e.g., RdBu), and categorical colormaps (e.g., tab20).



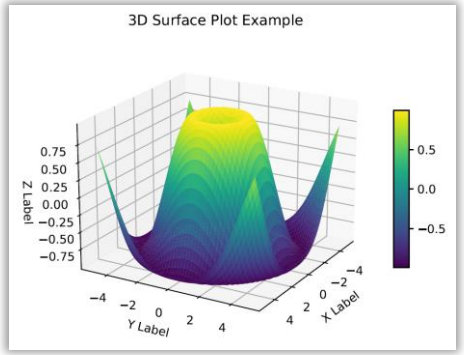

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate data for contour plot
x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
X, Y = np.meshgrid(x, y)
Z = X**2 + Y**2

# Create a Figure and an Axes (subplot)
fig, ax = plt.subplots()

# Create a contour plot with a colormap
contour = ax.contourf(X, Y, Z, levels=20, cmap='viridis')

# Add a colorbar
cbar = plt.colorbar(contour)

# Set labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Contour Plot with Colormap Example')

# Display the plot
plt.show()
```

The `levels` parameter specifies the number of contour levels, and the `cmap` parameter sets the colormap to `'viridis'`.

3. Matplotlib

3.6 Example 11: 3D plot

- You can plot in 3D using the Axes3D class, which can be acquired with the `fig.add_subplot(…, projection='3d')` method
- 3D plotting methods with the Axes3D object (ax):
  - 3D surface plotting: `ax.plot_surface()`
  - 3D Scatter plotting: `ax.scatter()`
  - 3D bar chart plotting: `ax.bar()`




```python
import numpy as np
import matplotlib.pyplot as plt

# Generate data
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create a Figure and 3D subplot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Create 3D surface plot
surface = ax.plot_surface(X, Y, Z, cmap='viridis')

# Set labels and title
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
ax.set_title('3D Surface Plot Example')

ax.view_init(elev=20, azim=30) # Adjust viewing angles

# Add colorbar
fig.colorbar(surface, ax=ax, shrink=0.5, aspect=10)

# Display the plot
plt.show()
```

The argument 111 can be broken down as follows:

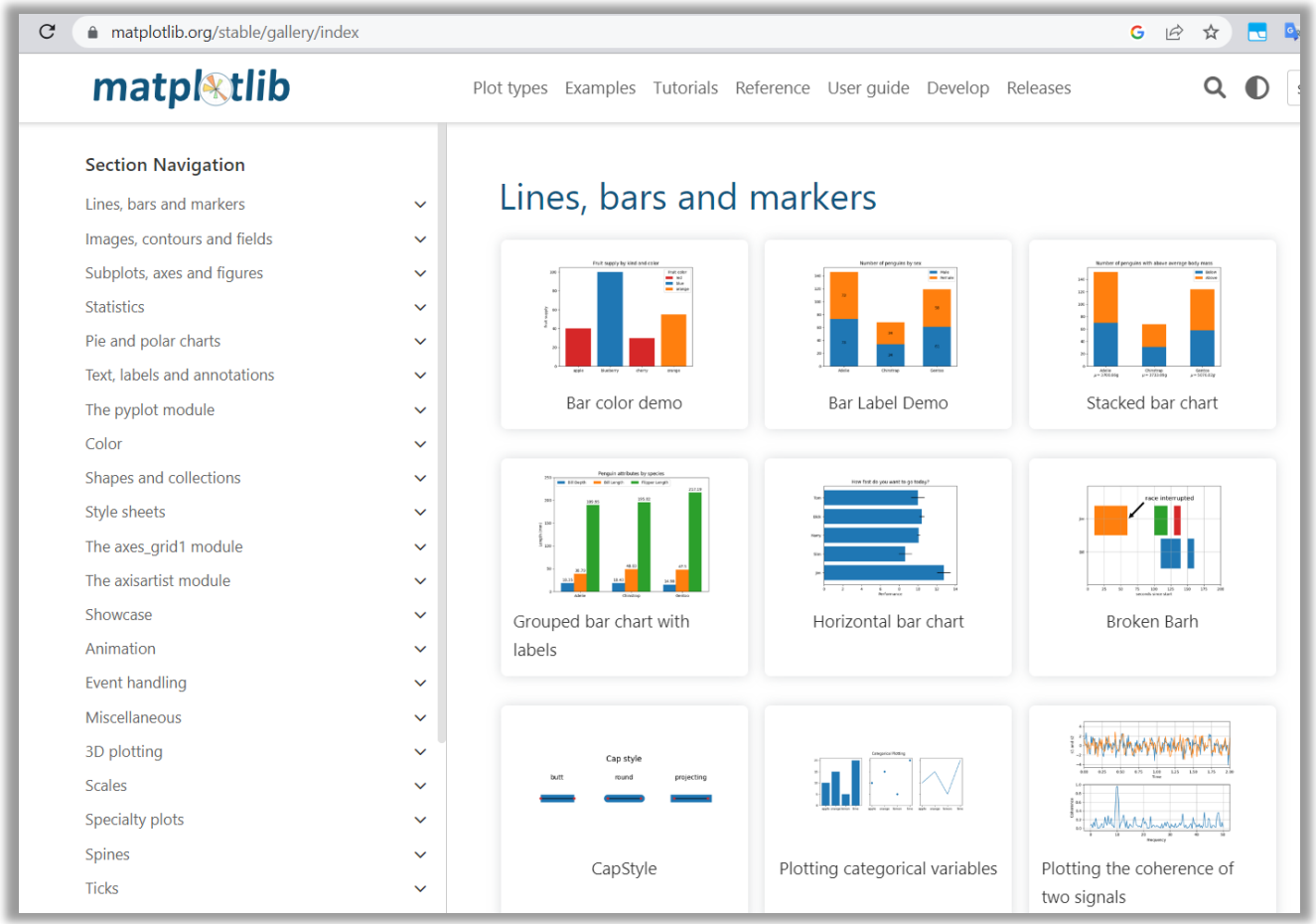
- The first digit (1) corresponds to the number of rows in the grid.
- The second digit (1) corresponds to the number of columns in the grid.
- The third digit (1) corresponds to the position of the subplot within the grid. Since this is the only subplot in the grid, it occupies the only available position.

Using add_subplot(111) or add_subplot(1, 1, 1) is a common way to create a single subplot that occupies the entire figure space

3. Matplotlib

3.6 Example: Need more?

You can check Matplotlib's example gallery to explore what kinds of plots meet your needs.
https://matplotlib.org/stable/gallery/index




THANK YOU
Happy Coding!